Pipeline-aware Scheduling of Polyhedral Process Networks

Christophe Alias, Julien Rudeau

Pipeline-aware Scheduling of Polyhedral Process Networks

Christophe Alias, Julien Rudeau

RESEARCH REPORT
N° 9314
December 2019
Project-Team Cash

ISSN 0249-6399 ISRN INRIA/RR--9314--FR+ENG

Pipeline-aware Scheduling of Polyhedral Process Networks

Christophe Alias
Laboratoire de l'Informatique du Parallélisme
CNRS, ENS de Lyon, Inria, UCBL, Université de Lyon
Lyon, France
Christophe.Alias@inria.fr

Julien Rudeau
Laboratoire de l'Informatique du Parallélisme
CNRS, ENS de Lyon, Inria, UCBL, Université de Lyon
Lyon, France
Julien.Rudeau@ens-lyon.fr

Abstract

The polyhedral model is a well known framework to develop accurate and optimal automatic parallelizers for high-performance computing kernels. It is progressively migrating to high-level synthesis through polyhedral process networks (PPN), a dataflow model of computation which serves as intermediate representation for high-level synthesis. Many locks must be overcome before having a fully working polyhedral HLS tool, both from a front-end (C → PPN) and back-end (PPN → FPGA) perspective. In this paper, we propose a front-end scheduling algorithm which reorganizes the computation of processes to maximize the pipeline efficiency of the processes' arithmetic operators. We show that our approach improve significantly the overall latency as well as the pipeline efficiency.

Keywords high-level synthesis, process networks, automatic parallelization, polyhedral model

1 Introduction

Since the end of Dennard scaling, energy efficiency (measured in flop/J) has become a major issue whenever the energy budget is limited, typically for embedded systems and high-performance computers (HPC). The current trend is to explore the trade-off between architecture programmability and energy efficiency (op/J). At the two extremes, an ASIC (application specific integrated circuit) finely tuned to realize a specific function is more energy efficient than a mainstream processor (Xeon, etc). Hence the rise of *hardware accelerators* [9] (Xeon-Phi, GPU, FPGA). FPGA appears as the solution: they combine the (potential) energy efficiency of a specialized circuit with the programmability. With FPGA, the program is a circuit configuration. However, designing a circuit is far more complex than writing a C program. Disruptive compiler technologies are required to generate automatically a circuit configuration from an algorithmic description (High-level synthesis, HLS) [6]. An HLS compiler must extract the parallelism of the input program and allocate the computation and the data to circuit resources (FPGA reconfigurable units). A crucial point is the choice of the intermediate representation. Dataflow models of computation appear as a good candidate, notably the polyhedral process networks (PPN) developed in the context of the Compaan project [12, 19] and addressed in this paper. A PPN is made of processes communicating through buffers with the KPN semantics [12]. The execution of a PPN is locally sequential (with a fixed schedule θ_P for each process P) and globally dataflow.

Floating point arithmetic operators ($+, -, \times, /, \sqrt{.}, \ldots$) used in hardware accelerators are pipelined to guarantee the circuit bandwidth. An operation i produces a result at the date $t(i) + \delta$, with δ the number of pipeline stages. If the result of i is used by an operation j, j will have to wait for its availability: $t(j) > t(i) + \delta$ (pipeline constraint). Otherwise, the pipeline of j will be frozen until the data is available. Scheduling under pipeline constraints – or *pipeline aware scheduling* – consists in reorganizing the computations to reduce the total execution time while satisfying the pipeline constraints. This problem is known to be NP-complete on simple sequential codes without tests and loops [4, 10] (basic blocks).

In this paper, we propose a *pipeline-aware scheduling algorithm* for the *front-end* (C → PPN) of an HLS compiler. Each process P executes multiple iterations with a pipelined datapath with δ_P stages. Our goal is to reorder the iterations executed by the processes to reduce the pipeline stalls and to improve the overall latency of the PPN. More specifically, we make the following contributions:

- We build on [1] to propose *a general pipeline-aware scheduling algorithm for polyhedral process networks*. The algorithm presented in [1] schedules perfect loop nests with uniform dependencies. Our extension covers the entire class of polyhedral process networks, with a far more general dependence and computation structure.
- We conduct a theoretical study of pipeline-aware scheduling, we define a notion of pipeline optimality and *we prove a necessary condition for a schedule to be optimal*. We show this condition to be sufficient under restricted conditions, corresponding to the precondition of [1]. As a side effect, *we prove the optimality of [1].*
- We applied our algorithm to the compute-intensive kernels of the polybench/C suite [15]. This show that, *in practice, our algorithm reduce significantly the overall latency and improve significantly the pipeline efficiency.*

The remainder of this paper is structured as follows. Section 2 introduces the polyhedral model and the polyhedral process networks. Section 3 provides a theoretical study of pipeline-aware scheduling of polyhedral process network, proves a necessary condition which inspires our algorithms and demonstrates the optimality of [1]. Section 4 recalls [1] and presents our pipeline-aware scheduling algorithm for PPN. Section 5 gives the experimental results obtained on the kernels from Polybench/C. Finally, Section 6 concludes this paper and draws future research directions.

2 Preliminaries

This section introduces the context of this paper. Section 2.1 presents the polyhedral model, a general framework to design automatic parallelizers, then Section 2.2 presents the Polyhedral process network, the HLS intermediate representation addressed in this paper.

2.1 Polyhedral model

The *polyhedral model* is a general framework to analyse and to transform programs. It abstracts loop iterations as a union of convex polyhedra – hence the name – and data accesses as affine functions. This way, precise – iteration-level – compiler algorithms may be designed (dependence analysis [7], scheduling [8] or loop tiling [5] to quote a few) leading to performances far beyond mainstream approaches.

Program model The polyhedral model manipulates loop kernels – referred to as *polyhedral programs* – which consist of nested `for` loops and `if` conditions manipulating arrays and scalar variables, that satisfies an *affinity* property: loop bounds, `if` conditions, and array access functions are *affine expressions* of surrounding loops counters and structure parameters. With polyhedral programs, the control is *static*: it only depends on the input size (the structure parameters), not the input values. This way, loop iterations and array accesses might represented statically. Polyhedral programs covers an important class of compute- and data-intensive loop kernels usually found in linear algebra and signal processing applications [3, 15]. With polyhedral programs, each iteration of a loop nest is uniquely represented by the vector of enclosing loop counters $\vec{i}$. The execution of a program statement S at iteration $\vec{i}$ is denoted by $\langle S, \vec{i}\rangle$. The set $\mathcal{D}_S$ of iteration vectors is called the *iteration domain* of S. Figure 1.(a) depicts a polyhedral program computing the composition of matrix-vector product $\vec{z} := A(B\vec{x})$, where A and B are two matrices and $\vec{x}$ is a vector. (b) depicts the iteration domains $\mathcal{D}_S$ and $\mathcal{D}_T$ (grey points) of statements S and T. Thanks to the affinity property, an iteration domain is always defined as a conjunction of affine constraints on the iteration vector $\vec{i}$ and *structure parameters* (here N, the matrix/vector size).

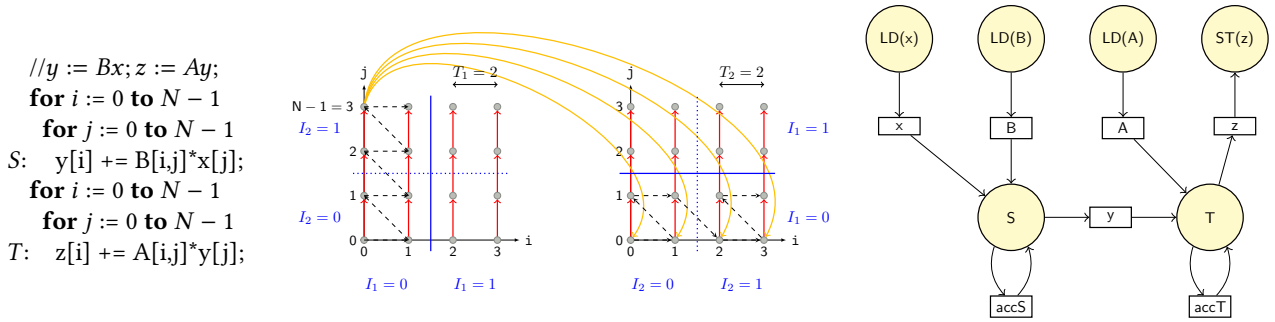
Data dependences In the polyhedral model, data dependences may be represented as Presburger relations $\{\langle S, \vec{i}\rangle \rightarrow \langle T, \vec{j}\rangle \mid \Phi(\vec{i}, \vec{j}, \vec{N})\}$, where $\Phi(\vec{i}, \vec{j}, \vec{N})$ is a formula of the Presburger arithmetic (usually a conjunction of affine constraints) whose variables are the source iteration $\vec{i}$, the target iteration $\vec{j}$ and the structure parameters $\vec{N}$. On our example, the *flow* data dependences might be summed up as:

$$(1)\ \{\langle S, i, j-1\rangle \rightarrow \langle S, i, j\rangle \mid 0 \leq i, j-1, j < N\}$$
$$(2)\ \{\langle S, i, N-1\rangle \rightarrow \langle T, 0, i\rangle \mid 0 \leq i < N\}$$
$$(3)\ \{\langle T, i, j-1\rangle \rightarrow \langle T, i, j\rangle \mid 0 \leq i, j-1, j < N\}$$

Data-dependences (1) (resp. (3)) are local to S (resp. T). They capture the `+=` accumulation across iterations of S (resp. T). They are represented with red arrows on (b). Data-dependences (2) captures the communication of $\vec{y}$ (carrying the intermediate matrix-vector product $B\vec{x}$) from S to T. They are depicted with yellow arrows on (b).

Scheduling In the polyhedral model, a program is parallelized by modifying the execution order (scheduling) and the allocation of data and computation across processing units. Schedules are *affine-per-statement*: for each statement S, an affine mapping θ_S is provided, which maps each execution $\langle S, \vec{i}\rangle$ to a timestamp $\theta_S(\vec{i}) = (t_1, \ldots, t_n) \in (\mathbb{Z}^n, \ll)$; the timestamps being order with the lexicographic ordering $\ll$: $(i, j) \ll (i', j')$ iff $i < i'$ or $i = i' \wedge j < j'$. Intuitively, $\theta_S(\vec{i})$ is the new iteration of $\langle S, \vec{i}\rangle$ in the target program. On our example, a schedule $\theta_S(i, j) = (0, i, j)$, $\theta_T(i, j) = (1, j, i)$ would execute all the iterations of S (coordinate 0), then all the iterations of T (coordinate 1). Also, the iterations of S are executed in the original order – (i, j): "for i for j"– while the iterations of T are executed as if loops i and j would be permuted – (j, i): "for j for i". A schedule θ induces a new execution order $\prec_\theta$. When $\prec_\theta$ is total, the schedule is said to be *sequential*. In that case, each execution $\langle S, \vec{i}\rangle$ has at most one successor $\langle S, \vec{j}\rangle$ in the execution sequence. The *control dependence* $\xrightarrow{ctl}$ relates $\langle S, \vec{i}\rangle$ to $\langle S, \vec{j}\rangle$: $\langle S, \vec{i}\rangle \xrightarrow{ctl} \langle S, \vec{j}\rangle$.

Loop tiling Loop tiling [5, 11] (also called loop blocking [20]) is a well known loop transformation to distribute the computation while optimizing the data locality. With loop tiling, iterations are grouped into *tiles*, then each tile is executed *atomically*: interdependence between tiles is forbidden. Figure 1.(b) gives an example of loop tiling in the polyhedral model. The iterations are simply partitioned by sliding cutting *hyperplanes* (depicted with blue lines – plain and dotted) defined by their equation at the origin. Iterations of S are cutted by the hyperplanes $\phi^S = (\phi_1^S, \phi_2^S)$ with $\phi_1^S(i, j) = i$ (blue, plain) and $\phi_2^S(i, j) = j$ (blue, dotted). The tile size is then defined as the cutting step in each direction. Here, we would have $(T_1, T_2) = (2, 2)$. After applying a loop tiling, an iteration $\vec{i}$ is transformed to $(\vec{I}, \vec{i})$, where $\vec{I}$ is the tile coordinate and $\vec{i}$ is an iteration belonging to the tile. For instance, the iterations of S would become $(\vec{I}, \vec{i}) = (I_1, I_2, i, j)$, see I_1 and I_2 on (b).



(a) Kernel (b) Pipeline-aware schedule (c) Polyhedral Process Network

Figure 1. Running example

2.2 Polyhedral process networks

Process networks [12, 19] are dataflow models of computation expressing naturally task-level parallelism for streaming applications. They are a relevant *intermediate representation* for parallelizing compilers, where a *front-end* extracts the parallelism and derive the process network, then a *back-end* maps the process network to the target architecture.

Polyhedral process networks (PPN) were primarily developed in the Compaan project [14, 16, 17, 19], to design high-level synthesis techniques leveraging the polyhedral model. With PPN, each statement is mapped to a *process*, then *buffers* are created to carry the flow of data between processes. PPN expose task parallelism, which may be tuned using process splitting [14] or partitioning [2]. The *semantics* of a PPN is *locally sequential, globally dataflow*: locally, a process S executes its iteration by following a schedule θ_S. When an iteration is ready to be executed, it tries to read the input data from the buffers. If all the inputs are available, the output value is computed and then written to the output buffers (a data may be written to several buffers). When all the writes are done, the next iteration is executed. To enforce the determinism of a PPN, there is exactly one buffer per couple (producer,read). On our example, we obtain the PPN depicted on Figure 1.(c). There are two kinds of processes: the compute processes translating the kernel itself (S and T) and the Load/Store processes responsible to interface the PPN with the outside world. Remark that the buffers follow exactly the flow dependences and that one buffer is created per couple (producer/read). Depending on the communication pattern [18], buffers might be implemented with a FIFO (when the data are read in the same order than they are written – *in-order*), a FIFO with a register (when data are read *in-order*, but multiple times) or a synchronized scratch-pad otherwise. Finally, each process S is assigned a polyhedral schedule θ_S. The derivation of a PPN from a polyhedral program leverages *direct* flow data-dependences [7] – flow data-dependences connecting the production of a value to its consumption (see the arrows on Figure 1.(b)). Note that a flow data-dependence might not be direct: for instance $\langle S, 0, 0\rangle \longrightarrow \langle S, 0, 2\rangle$ (not depicted). Formally, direct flow data-dependences are the *smallest subset of flow data-dependences* whose *transitive closure* is the set of flow data-dependences.

The goal of this paper is to schedule the processes (find the θ_S) so their pipelined datapath stalls as little as possible. On our example, processes S and T uses the multiply-accumulate arithmetic operator depicted on Figure 2 with a pipeline depth $\delta_S = \delta_T = m > 1$. Executing S in the original sequential order $\theta_S(i, j) = (i, j)$ would result in a pipeline stall at each iteration, since each iteration would wait for the result produced by the previous iteration. Now if the loops are simply permuted ($\theta_S(i, j) = (j, i)$), for each j, a sequence of *independent iterations* would be executed *without pipeline stall*. And when executing the next j, the result would be available, *provided the sequence of independent iterations takes more cycles than the multiply-accumulate pipeline depth*. However, this would require to store $N - m$ values out of the pipeline between the execution of two j. Hence, additional tuning is required. **The key idea developed in this paper is to hide the pipeline latency by scheduling the right number of independent operations.**

The next section provides a theoretical study for pipeline-aware scheduling and points out a necessary and sufficient condition for the existence of a stall-free schedule. Then section 4 will present our algorithm for pipeline-aware scheduling.

3 Pipeline-aware Scheduling

This section presents a theoretical study of pipeline-aware scheduling of PPN, which inspires the algorithm presented in the next section. In particular, we define a notion of *optimality* – a schedule is optimal iff it is pipeline freeze-free, and we show a *necessary condition* to reach this goal, which is proven to be *sufficient under more restricted hypothesis*.

Dataflow schedule, notion of optimality The execution of a PPN is *locally sequential* – with a schedule θ_P for each process P, and *globally dataflow*. To reason properly about

PPN, we need to define a *global dataflow schedule* assigning a global timestamp $t_P(\vec{i}) \in \mathbb{R}$ to each iteration $\vec{i}$ of each process P. In our model, each process P is assumed to be equipped with a pipelined datapath of latency δ_P: the execution of the iterations can be pipelined at each cycle and the result will be available δ_P cycles later:

Definition 3.1 (dataflow schedule)**.** *The dataflow schedule sets an execution date $t_T(\vec{j})$ for each iteration $\vec{j}$ of process T:*

- *If $\vec{j}$ is the very first iteration of T and if it does not wait for any input (no predecessor w.r.t. the dependence relation $\rightarrow$, then $t_T(\vec{i})$ is set to* 0.
- *Otherwise, consider the predecessors of $\langle T, \vec{j}\rangle$ for the* dependence relation $\rightarrow$*, $\langle S_1, \vec{i}_1\rangle, \ldots, \langle S_p, \vec{i}_p\rangle$, and for the* control relation $\xrightarrow{ctl}$*, $\langle T, \vec{i}\rangle$. We define:*

$$t_T(\vec{j}) = \max\{t_{S_1}(\vec{i}_1) + \delta_{S_1}, \ldots, t_{S_p}(\vec{i}_p) + \delta_{S_p}, t_T(\vec{i}) + 1\}$$

In other words, $\langle T, \vec{j}\rangle$ is executed as soon as possible *once it has control and once incoming dependences are satisfied.*

The maximum term in $\max\{t_{S_1}(\vec{i}_1) + \delta_{S_1}, \ldots, t_{S_p}(\vec{i}_p) + \delta_{S_p}, t_T(\vec{i}) + 1\}$ defines the *winning dependence*: if it is $t_T(\vec{i}) + 1$, the control dependence $\langle T, \vec{i}\rangle \xrightarrow{ctl} \langle T, \vec{j}\rangle$ *wins over the data-dependence*. This means that iteration $\vec{j}$ of process T might be executed directly without waiting for input data. *A contrario*, when a data-dependence wins, the iteration has to wait until the data is available. This observation leads to the following notion of optimality:

Definition 3.2 (Optimality criterion)**.** *A dataflow schedule t is* optimal *iff process iterations never wait for an incoming data:*

$$\langle S, \vec{i}\rangle \xrightarrow{ctl} \langle S, \vec{j}\rangle \Rightarrow t_S(\vec{j}) = t_S(\vec{i}) + 1$$

A necessary condition for optimality As pointed out, a good schedule needs to hide the pipeline latency by executing independent iterations between the source and the target of a data dependence. Since we deal with process networks, we need to consider *dependence chains across processes*:

Definition 3.3 (Latency)**.** *The* latency *of a dependence chain $c : \omega_1 \rightarrow \ldots \rightarrow \omega_{n-1} \rightarrow \omega_n$ where $n \geq 2$ is:*

$$\lambda(c) = t(\omega_{n-1}) - t(\omega_1) + \delta_{\omega_{n-1}}$$

In particular, when the chain is restricted to a single dependence ($n = 2$), we consider the latency of the source iteration: $\lambda(c) = \delta_{\omega_1}$. While the dependence chain is completed, a certain number of independent iterations must be executed. Hence the notion of *distance*:

Definition 3.4 (Distance)**.** *The* distance *between two iterations $\vec{i}$ and $\vec{j}$ of process S is the number of iterations of S executed between $\vec{i}$ and $\vec{j}$:*

$$\Delta_S(\vec{i}, \vec{j}) = \text{card}\{t_S(\vec{k}) \mid t_S(\vec{i}) \leq t_S(\vec{k}) < t_S(\vec{j})\}$$

We focus on the case where $\langle S, \vec{i}\rangle \xrightarrow{d} \langle S, \vec{j}\rangle$. In that case, $\Delta_S(\vec{i}, \vec{j})$ is called the *dependence distance*. We can now define our optimality condition:

Definition 3.5 (Condition (C))**.** *The condition (C) is defined as follows: For each dependence chain $c : \langle S, \vec{i}\rangle \rightarrow \ldots \rightarrow \langle S, \vec{j}\rangle$:*

$$\Delta_S(\vec{i}, \vec{j}) \geq \lambda(c)$$

This expresses whatever the dependence chain c from iteration $\vec{i}$ to $\vec{j}$ of process S: meanwhile it is complete, the process S never waits, since it has more iterations to execute ($\Delta_S(\vec{i}, \vec{j})$) than the latency of c ($\lambda(c)$). We now prove the main result.

Theorem 3.1. *(C) is a necessary condition for the optimality of t:*

$$t \text{ is optimal} \Longrightarrow (C)$$

Proof. Consider a data dependence chain $c : \langle S, \vec{i}\rangle \rightarrow \omega_1 \ldots \omega_{n-1} \rightarrow \langle S, \vec{j}\rangle$ and the control chain relating each side of c in S: $\langle S, \vec{i}\rangle \xrightarrow{ctl} \langle S, \vec{i}_1\rangle \xrightarrow{ctl} \ldots \xrightarrow{ctl} \langle S, \vec{i}_{\ell-1}\rangle \xrightarrow{ctl} \langle S, \vec{j}\rangle$. By definition, $\Delta_S(\vec{i}, \vec{j}) = \ell$. By hypothesis, t is optimal. Hence, $t_S(\vec{i}_{\ell-1})+1 = t_S(\vec{i}_{\ell-2})+2 = \ldots = t_S(\vec{i})+\ell$. Hence: $t_S(\vec{i}_{\ell-1})+1 = t_S(\vec{i}) + \Delta_S(\vec{i}, \vec{j})$.

Also, since t is optimal, the control dependence $\langle S, \vec{i}_{\ell-1}\rangle \xrightarrow{ctl} \langle S, \vec{j}\rangle$ wins over the data dependence $\omega_{n-1} \rightarrow \langle S, \vec{j}\rangle$, we have $t_S(\vec{i}_{\ell-1}) + 1 \geq t(\omega_{n-1}) + \delta_{\omega_{n-1}}$. Hence: $t_S(\vec{i}) + \Delta_S(\vec{i}, \vec{j}) \geq t(\omega_{n-1}) + \delta_{\omega_{n-1}} = t(\omega_{n-1}) - t_S(\vec{i}) + t_S(\vec{i}) + \delta_{\omega_{n-1}}$. Subtracting $t_S(\vec{i})$ from each member, we obtain: $\Delta_S(\vec{i}, \vec{j}) \geq t(\omega_{n-1}) - t_S(\vec{i}) + \delta_{\omega_{n-1}} = \lambda(c)$. Hence, (C) holds. □

□

The converse is not true in general, it is possible to exhibit a counter example which verifies (C) while waiting for input data. We suspect that most of these examples can be transformed (by compacting and shifting the schedule) to be optimal.

Sufficiency of (C) We now show that (C) is sufficient under restricted conditions. The following lemma states that, when all the incoming data-dependences are *local to the process* (source iteration and target iteration belongs to the process), the control dependence always wins under (C):

Lemma 3.2. *Consider an operation $\langle S, \vec{j}\rangle$ with a control dependence from $\langle S, \vec{i}'\rangle$: $\langle S, \vec{i}'\rangle \xrightarrow{ctl} \langle S, \vec{j}\rangle$ and a data dependence from $\langle S, \vec{i}\rangle$: $\langle S, \vec{i}\rangle \xrightarrow{d} \langle S, \vec{j}\rangle$. If (C), then the control dependence wins:*

$$t_S(\vec{i}') + 1 \geq t_S(\vec{i}) + \delta_S$$

Proof. Starting from $(C) : \Delta_S(\vec{i}, \vec{j}) \geq \lambda(d) = \delta_S$ (*), we obtain $\text{card}\{t_S(\vec{k}) \mid t_S(\vec{i}) \leq t_S(\vec{k}) < t_S(\vec{j})\} \geq \delta_S$ (definition Δ_S). Let $\langle S, \vec{i}\rangle \xrightarrow{ctl} \langle S, \vec{i}_1\rangle \ldots \xrightarrow{ctl} \langle S, \vec{i}_{\ell-1}\rangle = \langle S, \vec{i}'\rangle \xrightarrow{ctl} \langle S, \vec{j}\rangle$ be the iterations of S with dates $t_S(\vec{k})$ sorted in the control order.

By definition of t_S, $t_S(\vec{i'}) + 1 \geq t_S(\vec{i}) + \ell = t_S(\vec{i}) + \Delta_S(\vec{i}, \vec{j})$. In turn, $t_S(\vec{i}) + \Delta_S(\vec{i}, \vec{j}) \geq t_S(\vec{i}) + \delta_S$ (using (*)). Hence the result: $t_S(\vec{i'}) + 1 \geq t_S(\vec{i}) + \delta_S$. □ □

This implies that a pool of synchronization-free processes may be scheduled optimally provided (C):

Corollary 3.3. *If no data-dependence hold from T to S, $S \neq T$:*

$$t \text{ is optimal} \Longleftrightarrow (C)$$

Proof. The necessary part $\Rightarrow$ is a direct sub-case of theorem 3.1. To show the sufficient part $\Leftarrow$, consider an operation $\langle S, \vec{j} \rangle$ to be scheduled, all the incoming data dependences: $\langle T_m, \vec{i}_m \rangle \rightarrow \langle S, \vec{j} \rangle$ for $1 \leq m \leq k$ and the incoming control dependence: $\langle S, \vec{i} \rangle \xrightarrow{ctl} \langle S, \vec{j} \rangle$. By hypothesis, data dependences are local to processes: $T_m = S$ for any $1 \leq m \leq k$. By Lemma 3.2, the control dependence wins: $t_S(\vec{i})+1 \geq t_{T_m}(\vec{i}_m)+\delta_{T_m}$ for any $1 \leq m \leq k$. Hence, $\langle S, \vec{i} \rangle \xrightarrow{ctl} \langle S, \vec{j} \rangle \Rightarrow t_S(\vec{j}) = t_S(\vec{i}) + 1$. This proves the optimality of t □. □

4 Our Algorithm

We build on the algorithm presented in [1] to define our pipeline-aware scheduling algorithm. The algorithm presented in [1] derives a pipeline-aware schedule for a *perfect loop nest* with *uniform dependences*. Conceptually, this may be view as a PPN with a single process and a restricted communication pattern. The challenge is to extend this algorithm to a PPN with communicating processes.

Starting from a perfect loop nest, [1] relies on a loop tiling with a *concurrent starting hyperplane* to schedule a number of independent iteration between the source and the target of a dependence. On Figure 1.(b), restricted to the domain of S, this algorithm would find the tiling depicted with blue lines (plain and dotted) and would size the tile (size T_1) to tune the dependence distance so $\Delta_S(i, j-1, i, j) \geq \delta_S$ (to ease the presentation, we choose $\delta_S = 2$). Here, the concurrent starting hyperplane is given by the blue, dotted line ($\tau_2 = (0, 1)$): Along that plane, all the iterations may be executed in parallel. In particular, they are independent and may feed the pipeline between the source and the target of a dependence. This way, our algorithm would find the polyhedral schedule $\theta(I_1, I_2, i, j) = (I_1, I_2, j, i)$. The schedule derived by [1] always verifies (C), hence *this algorithm always find an optimal schedule, according to Corollary 3.3.*

We extend this algorithm to schedule the processes of a PPN. The main challenge is to enforce condition (C): how to make sure that the dependence distance will be bigger that the dependence chain latency, whatever the dependence chain? How to deal with general dependence, not necessarily uniform as considered in [1]? We propose the extension depicted in Algorithm 1. First, we build a *global tiling* (1) minimizing the dependence distance, by applying the algorithm described in [5]. The result is a global tiling and schedule, which tends to *execute a dependence target as close as possible to its source*. This way, a process will tend to read a data as soon as possible after its production. This has the effect of reducing the global latency: on 1.(b), the bands separated by a thick blue line would be executed in a pipelined fashion: first S, left band, then in parallel: S right band and T bottom band, finally: T top band. The remainder of our algorithm is as follows: for each process, we select a concurrent plane, or we build it if it does not exists (step 7). Finally, we prescribe an execution along the concurring hyperplane (step 8), similarly as [1]. Although Theorem 3.1 tells us that condition (C) is necessary, we point out that (C) is not sufficient in general. Hence, there is *a priori* no guarantee that our algorithm will reach an optimal schedule, when it exists. In practice, our algorithm improves significantly the pipeline efficiency of PPN, as show in the next section.

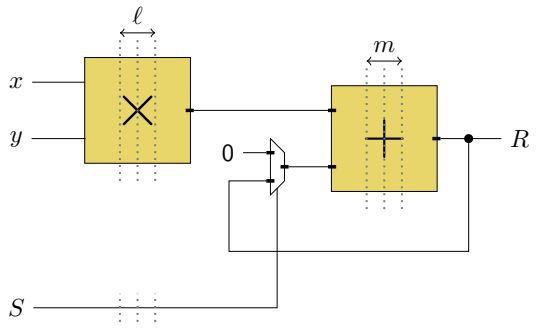


Figure 2. Pipelined multiply-accumulate operator

Algorithm 1: Pipeline-aware scheduling algorithm, Polyhedral process network

Input : A polyhedral process network
Output: A pipeline-aware schedule *affine-per-process* θ

1 Find a *global tiling* $\phi^S := (\phi^S_1, \ldots, \phi^S_n)$ *minimizing the dependences distances*, for each process S
2 **foreach** *process* S **do**
3 **if** ϕ^S *has a concurrent start hyperplane* ϕ^S_k **then**
4 Swap ϕ^S_k and ϕ^S_n
5 **else**
6 Substitute ϕ^S_n with $\phi^S_1 + \ldots + \phi^S_n$
7 **end**
8 Set the process schedule
   $\theta_S(\vec{I}, \vec{i}) := (\vec{I}, \phi^S_1(\vec{i}), \ldots, \phi^S_{n-2}(\vec{i}), \phi^S_n(\vec{i}), \phi^S_{n-1}(\vec{i}))$
9 **end**

On our example, our algorithm would find the tiling hyperplanes:

$$\phi^S_1(i,j) = i \quad \phi^S_2(i,j) = j$$
$$\phi^T_1(i,j) = j \quad \phi^T_2(i,j) = i$$

Tiling hyperplane ϕ^S_2 (resp. ϕ^T_1) has a concurring start on S (resp. T), its represented with dotted line on $\mathcal{D}_S$. Hence, our

algorithms produces the schedules $\theta_S(I_1, I_2, i, j) = (I_1, I_2, j, i)$ and $\theta_T(I_1, I_2, i, j) = (I_1, I_2, j, i)$, which happens to be the same here. The tile size T_1 is set to the pipeline depth m to enforce (C). The order prescribed by θ_S and θ_T is depicted with dashed black arrows. This schedule is optimal: once S and T are started, they will not have to wait for an input data. Note that the very first iteration of T have to wait for S to be started. This fits with our definition of optimality which concerns all the iterations of a process but the first one.

5 Experimental Evaluation

This section presents the experimental results obtained on the benchmarks of the polyhedral community. We show that, in practice, our algorithm reduce significantly the overall PPN latency while improving the pipeline efficiency.

Experimental Setup We have run our algorithm on the kernels of PolyBench/C v3.2 [15], a benchmark suite with compute-intensive linear algebra kernels from the polyhedral model community. For each kernel, we generate a polyhedral process network using our DCC compiler. DCC applies a dataflow analysis, an array expansion, and then restructures the communications to comply with the PPN channel policy (one channel per couple producer/read) as described in [17]. Then, the channels are typed depending on the communication pattern (FIFO, etc) and sized as explained in Section 2.

We designed a lightweight simulator which, given an execution trace for each process, computes the dataflow schedule t (as defined in Section 3) and deduces the global latency of the PPN. The execution trace is provided for a fixed value of input size (referred to as structure parameter $\vec{N}$ in Section 2), generally the input matrices/vectors size. For each kernel, we analyze the trace obtained with the original execution order (*base*) and the trace obtained after applying our pipeline-aware scheduling (*ours*). Finally, we compute the *pipeline efficiency* for each process P with:

$$\text{eff}_P = 1 - \frac{\text{bubbles}_P}{t^P_{\max} - t^P_{\min} + \delta_P}$$

Where bubbles is the total number of *waiting cycles*: given the sequence of dates $t^P_{\min} = t_1, \ldots, t_\ell = t^P_{\max}$ obtained for P, bubbles is simply the sum of "gaps" $\sum_{i>0} t_i - t_{i-1} + 1$. The global efficiency is then defined by the average of the eff_P for each process P, ponderated by its number of iterations.

The results are depicted in Table 1 and Figures 3.(a) and (b). Figure (a) gives the latency obtained after scheduling the PPN with our algorithm. The latency is *normalized*, i.e. expressed an percent of base latency – the smaller, the better. Finally, Figure (b) present the pipeline efficiency obtained on the base PPN (blue bar) and the PPN optimized with our algorithm (yellow bar) – the bigger, the better.

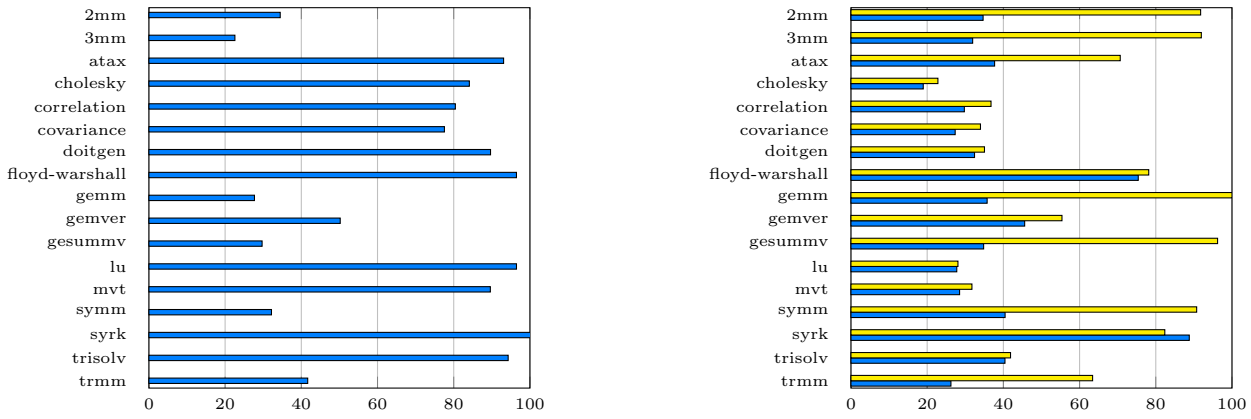
Results Globally, our algorithm succeeds to reduce significantly the overall latency and to improve the pipeline efficiently. In average, the latency is reduced by 32% and the pipeline efficiency is improved by 30%. Not surprisingly, the optimal is almost never reached except for the `gemm` kernel. `gemm` is a classical BLAS [13] function, that computes $C := \alpha AB + \beta C$ where capital letter denotes matrices and Greek letters denotes constants. Its PPN consists of two process: one process computing βC, which forwards its result to a process computing αAB and the final summation. We are in the same case as our motivating example, with forward dependences (no cycles between processes), and several parallel accumulation (one per element of C) where an orthogonal tiling composed with a loop permutation allow to reach an optimal schedule. On `syrk`, the performances are downgraded. This due to the structure of the iteration domain, which is triangular. The base version was already optimized, and performs well until the upper vertex of the triangle is reached, causing pipeline stalls since then the dependence distance become smaller than the pipeline depth. On the tiled version, there is no one, but several such small triangles on the borders of the iteration domain (due to the tiling), each triangle reproducing the same pipeline stalls.

Kernel	Latency (cycles)			Pipeline efficiency (%)		
	Base	**Ours**	Gain	Base	**Ours**	Gain
2mm	2066	712	66%	35	92	62%
3mm	3487	787	77%	32	92	65%
atax	246	229	7%	38	71	47%
cholesky	239	201	16%	19	23	17%
correlation	1124	904	20%	30	37	19%
covariance	1319	1023	22%	27	34	20%
doitgen	14855	13319	10%	32	35	7%
floyd-warshall	679	655	43%	75	78	4%
gemm	1859	515	72%	36	100	64%
gemver	474	238	50%	46	55	18%
gesummv	239	71	70%	35	96	64%
lu	255	246	4%	28	28	1%
mvt	231	207	10%	29	32	10%
symm	743	239	68%	40	91	55%
syrk	333	363	-9%	89	82	-8%
trisolv	105	99	61%	40	42	3%
trmm	739	308	58%	26	63	59%

Table 1. Detailed results

6 Conclusion

In this paper, we have proposed an algorithm for pipeline-aware scheduling of a polyhedral process network. This is, as far as we know, the first algorithm ever to solve this problem on polyhedral process networks. With our algorithm the pipeline stalls are reduced and the process tends to be executed without waiting for input data. We also developed a theoretical study of pipeline-aware scheduling and we prove a necessary condition (C) on the schedule so the optimal execution is reached. We also prove this condition to be sufficient on a restricted case, corresponding to the algorithm



(a) Optimized latency (%base)

(b) Pipeline efficiency (%): ours=yellow, base=blue

Figure 3. Latency and pipeline efficiency improvement

presented in [1], that we show to be optimal. Experimental results show that, in practice, our algorithm reduces the overall latency of a PPN and improve significantly the pipeline efficiency of processes.

In the future, we plan to extend our theoretical study with a less restricted hypothesis on the PPN structure so (C) becomes sufficient. In some cases, a non-optimal schedule verifying (C) may be transformed to an optimal schedule with date shifting and compaction. We plan to identify formally such cases and to mechanize this transformation.

References

[1] Christophe Alias, Bogdan Pasca, and Alexandru Plesco. 2012. FPGA-Specific Synthesis of Loop-Nests with Pipeline Computational Cores. *Microprocessors and Microsystems* 36, 8 (November 2012), 606–619.

[2] Christophe Alias and Alexandru Plesco. 2015. *Data-aware Process Networks.* Research Report RR-8735. Inria - Research Centre Grenoble – Rhône-Alpes. 32 pages.

[3] C. Bastoul, A. Cohen, S. Girbal, S. Sharma, and O. Temam. 2003. Putting polyhedral loop transformations to work. In *International Workshop on Languages and Compilers for Parallel Computing (LCPC).*

[4] David Bernstein, Michael Rodeh, and Izidor Gertner. 1989. On the complexity of scheduling problems for parallel/pipelined machines. *IEEE Trans. Comput.* 38, 9 (1989), 1308–1313.

[5] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Program Optimization System. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).*

[6] Philippe Coussy and Adam Morawiec. 2008. *High-Level Synthesis: From Algorithm to Digital Circuit.* Springer.

[7] Paul Feautrier. 1991. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming* 20, 1 (1991), 23–53. https://doi.org/10.1007/BF01407931

[8] Paul Feautrier. 1992. Some Efficient Solutions to the Affine Scheduling Problem, Part II: Multi-Dimensional Time. *International Journal of Parallel Programming* 21, 6 (Dec. 1992), 389–420.

[9] Al Geist and Daniel A Reed. 2015. A survey of high-performance computing scaling challenges. *International Journal of High Performance Computing Applications* (2015), 1094342015597083.

[10] John L Hennessy and Thomas Gross. 1983. Postpass code optimization of pipeline constraints. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 5, 3 (1983), 422–448.

[11] F. Irigoin and R. Triolet. 1988. Supernode Partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'88).* 319–329. https://doi.org/10.1145/73560.73588

[12] Gilles Kahn. 1974. The Semantics of Simple Language for Parallel Programming. In *IFIP Congress 74.* 471–475.

[13] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. 1979. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Trans. Math. Softw.* 5, 3 (Sept. 1979), 308–323.

[14] Sjoerd Meijer. 2010. *Transformations for Polyhedral Process Networks.* Ph.D. Dissertation. Leiden Institute of Advanced Computer Science (LIACS), and Leiden Embedded Research Center, Faculty of Science, Leiden University.

[15] Louis-Noël Pouchet. 2012. Polybench: The polyhedral benchmark suite. *URL: http://www. cs. ucla. edu/~ pouchet/software/polybench/[cited July,]* (2012).

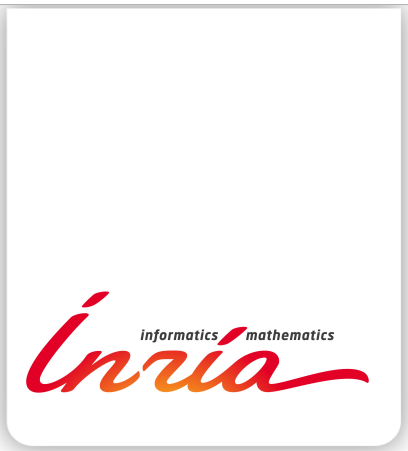
[16] Edwin Rijpkema, Ed F Deprettere, and Bart Kienhuis. 2000. Deriving process networks from nested loop algorithms. *Parallel Processing Letters* 10, 02n03 (2000), 165–176.

[17] Alexandru Turjan. 2007. *Compiling nested loop programs to process networks.* Ph.D. Dissertation. Leiden Institute of Advanced Computer Science (LIACS), and Leiden Embedded Research Center, Faculty of Science, Leiden University.

[18] Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. 2007. Classifying interprocess communication in process network representation of nested-loop programs. *ACM Transactions on Embedded Computing Systems (TECS)* 6, 2 (2007), 13.

[19] Sven Verdoolaege. 2010. *Polyhedral Process Networks.* 931–965.

[20] Michael Wolfe. 1989. Iteration Space Tiling for Memory Hierarchies. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 357–361. http://dl.acm.org/citation.cfm?id=645818.669220



Pipeline-aware Scheduling of Polyhedral Process Networks

Christophe Alias*, Julien Rudeau†

Project-Team Cash

Research Report n° 9314 — December 2019 — 7 pages

Abstract: The polyhedral model is a well known framework to develop accurate and optimal automatic parallelizers for high-performance computing kernels. It is progressively migrating to high-level synthesis through polyhedral process networks (PPN), a dataflow model of computation which serves as intermediate representation for high-level synthesis. Many locks must be overcome before having a fully working polyhedral HLS tool, both from a front-end (C $\rightarrow$ PPN) and back-end (PPN $\rightarrow$ FPGA) perspective. In this report, we propose a front-end scheduling algorithm which reorganizes the computation of processes to maximize the pipeline efficiency of the processes' arithmetic operators. We show that our approach improve significantly the overall latency as well as the pipeline efficiency.

Key-words: High-level synthesis, local scheduling, pipeline efficiency, polyhedral process networks

* Inria/ENS-Lyon/UCBL/CNRS
† Inria/ENS-Lyon/UCBL/CNRS

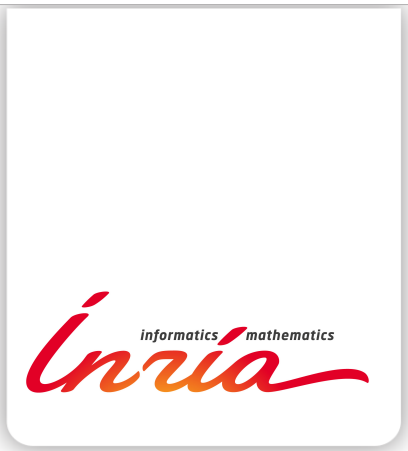
RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Pipeline-aware Scheduling of Polyhedral Process Networks

Résumé : Le modèle polyédrique est un *framework* mature pour développer des parallléliseurs automatiques précis et efficaces pour le calcul haute-performance. Il migre progressivement vers la synthèse de circuits haut-niveau (High-Level Synthesis, HLS) à travers le formalisme des *Polyhedral Process Networks* (PPN), un modèle de calcul dataflow qui sert de représentation intermédiaire pour la synthèse de circuits haut-niveau. Beaucoup de verrous doivent encore être résolus avant d'avoir un outil de HLS polyédrique opérationel, autant sur les aspects *front-end* (C $\rightarrow$ PPN) que *back-end* (PPN $\rightarrow$ FPGA). Dans ce rapport, nous proposons un algorithme d'ordonnancement *front-end* qui réorganise le calcul des processus de façon à améliorer l'efficacité du pipeline du chemin de données pour chacun des processus. Les résultats expérimentaux montrent une amélioration significative de la latence globale du circuit et de l'efficacité moyenne des pipelines.

Mots-clés : Synthèse de circuits haut-niveau, ordonnancement, pipeline, polyhedral process networks



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399